Ryan C. Dallavia

CS-242 Summer 2018

07/22/2018

1. To ensure all items are hashed to a location in our table, we must use a hash function that contemplates the size of the underlying array on which our table is built. We do this by setting the index in the hash table to which an item will be hash to the value of the hash function modulo the length of the table. Prior to rehashing, our table length is 9. Modulo exhibits a circular property. If we “mod” the integers 1 through 9, sequentially, by 9 (e.g., 1%9, 2%9 . . . 9%9 etc.), our function will return the series 1, 2, 3 . . .7, 8, 0, all representing the remainders of dividing each number by 9. If we move to the next set of 9 integers, the cycle repeats itself. If we feed items, sequentially, into our hash function our hash function “hashes” in a manner not unlike the cycle produced by modulo. If the number of items to be hashed, exceeds the number of “buckets” in the underlying array, this implies there will be a great many collisions to resolve on a regular basis and the chains used to resolve each collision will be length.

By contrast, if we hash random values to our table, we escape (to an extent) the predictability of the modulo cycle. While the function produces the same values, they do not appear in the same order, rather they are randomized.

A defining characteristic of a “good” hash function is its ability to produce randomized values that are evenly distributed across the underlying data structure. Sending sequential data into your hash function may not necessarily hobble a function’s ability to do this, but from an elapsed time perspective, it seems to pay off when you cooperate with the function by loading randomized, as opposed to sequentially ordered, inputs.

1. The data produced were inconsistent and, at times, counterintuitive. Loading the table with the integers 1-10 and searching for 3 should take longer than loading the table with integers 1-3 and searching for 3. When the load factor (i.e., the number of occupied buckets relative to number of total available buckets) is high, the likelihood of collision increases. More collisions imply, at least under chaining, a more extensive search process for a given target. In our trial, however, the data were fairly inconsistent with the fill and lookup in the overloaded table producing, at times, slightly faster speeds than the table with the low load factor. This may be the result of other processes running in the background, as well as machine specific factors.
2. Once again, the data produced were inconsistent and, at times, counterintuitive. Loading the table with the integers 1-10, removing 4-10, and searching for 3 should take longer than loading the table with integers 1-3, removing 1 and 2, and searching for 3. One would think the removal on *n* items would add to the running time of any routine, whether or not the removal process is linear. Thus, one would expect the routine in which more numbers are removed to take longer comparable routines in which fewer items are removed from the table. Furthermore, one would expect a lookup in a table with the lower load factor to be relatively faster than that occurring in a table with a comparatively higher one. Two possible explanations are 1) this may be the result of other processes running in the background, as well as machine specific factors and/or 2) there may be a range of load factors within which performance improvements are negligible.